

Lecture 18

What we've done and what's to come

Announcements

- Final exam Friday
 - This room (Hewlett 200)
 - 3:30 – 6:30pm, Friday 6/9
 - You may bring two double-sided sheets of notes
 - **The final will be harder than the midterm**
- Resources
 - Practice exams online
 - Office hours
 - Book, notes, slides, HW, piazza
- Course evaluations open
 - Please fill them out!

Today

- What just happened?
 - A whirlwind tour of CS161



- What's next?
 - A few gems from future algorithms classes



It's been a fun ride...

Sorting and
friends!

Data structures:
BSTs and Hashing!

Graphs!

Dynamic
Programming!

Greedy algorithms!

Scheduling
and etc.

MinCuts and
MaxFlows

Divide-and-conquer
and recurrence
relations

$O()$ and
worst-case
analysis

Randomized
algorithms

BFS, DFS, SCCs

LCS, Knapsack(s)

Bellman-Ford,
Floyd-Warshall

MSTs: Prim
and Kruskal

Karger, Ford-
Fulkerson

Dijkstra's algorithm



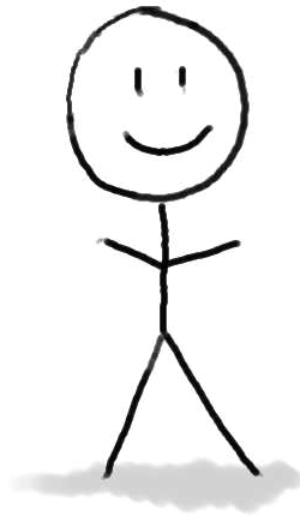
What have we learned?

17 lectures in 12 slides.

General approach to algorithm design and analysis

Can I do better?

To answer this question we need
both **rigor** and **intuition**:



Algorithm designer



Plucky the
Pedantic Penguin

Detail-oriented
Precise
Rigorous



Lucky the
Lackadaisical Lemur

Big-picture
Intuitive
Hand-wavey

To even begin to analyze algorithms, we need a model

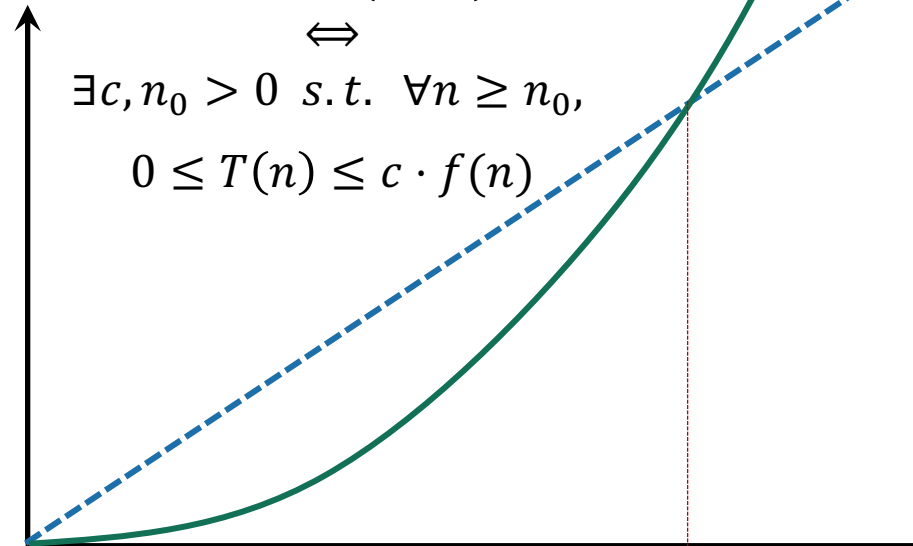
- What does it mean for an algorithm to succeed?
- What does it mean for an algorithm to be fast?

Worst-case analysis

HERE IS AN
INPUT!

big-Oh notation

$$\begin{aligned} T(n) &= O(f(n)) \\ &\Leftrightarrow \\ \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \\ 0 \leq T(n) &\leq c \cdot f(n) \end{aligned}$$



Algorithm design paradigm: divide and conquer

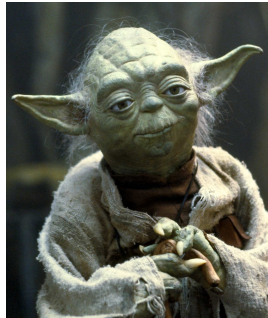
- Like MergeSort!
- Or Karatsuba's algorithm!
- Or SELECT!
- How do we analyze these?

By careful
analysis!

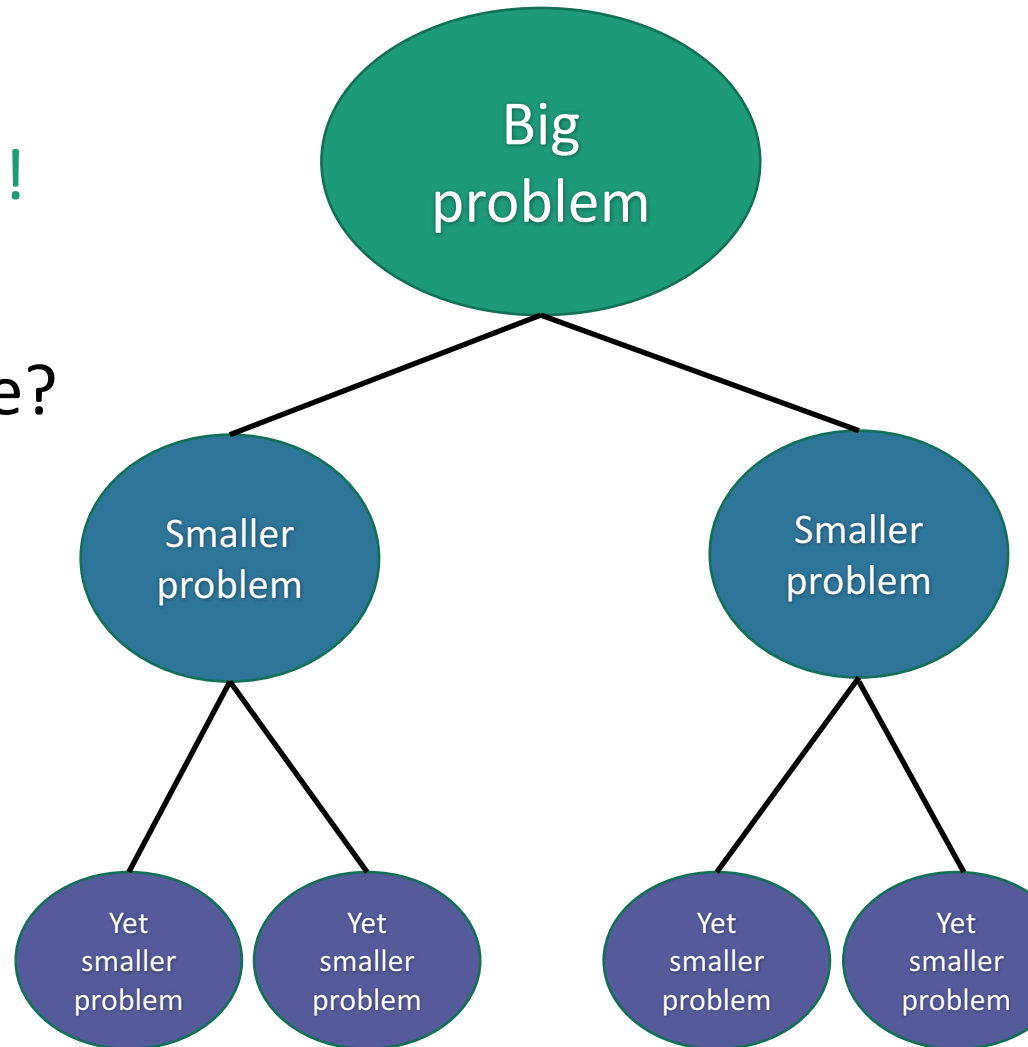


Plucky the
Pedantic Penguin

Useful shortcut, the
master method is.



Jedi master Yoda



While we're on the topic of sorting

Why not use randomness?

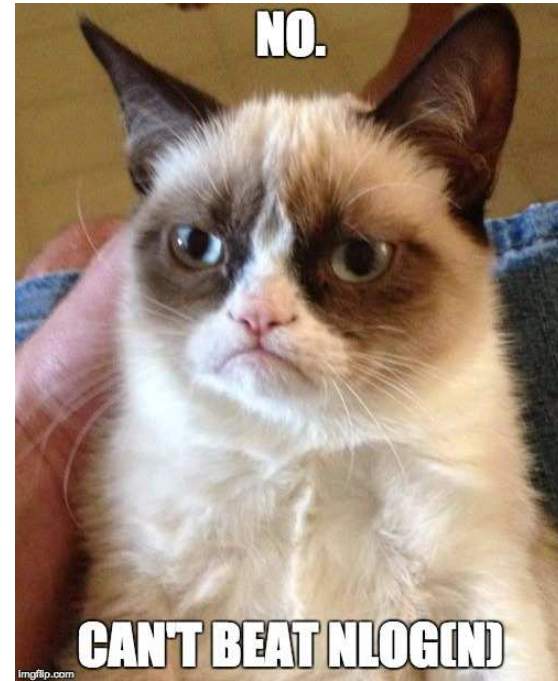
- We analyzed **QuickSort!**
- Still **worst-case input**, but we use **randomness** after the input is chosen.
- Always correct, usually fast.
 - This is a Las Vegas algorithm



All this sorting is making me wonder...

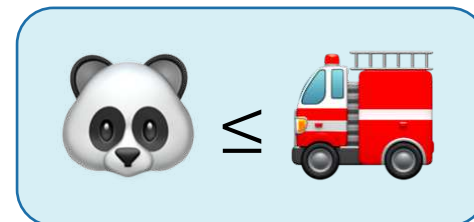
Can we do better?

- Depends on who you ask:



- **RadixSort** takes time $O(n)$ if the objects are, for example, small integers!

- Can't do better in a **comparison-based** model.



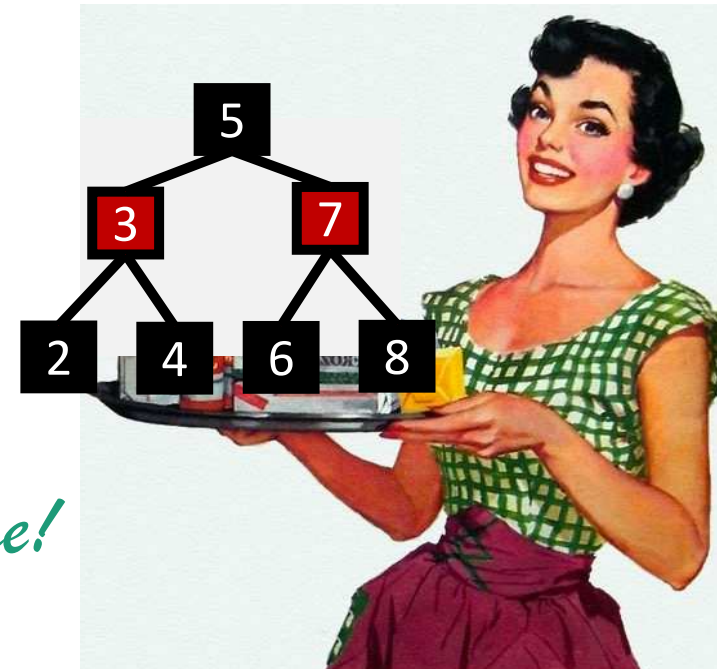
beyond sorted arrays/linked lists: Binary Search Trees!

- Useful data structure!
- Especially the self-balancing ones!

Red-Black tree!

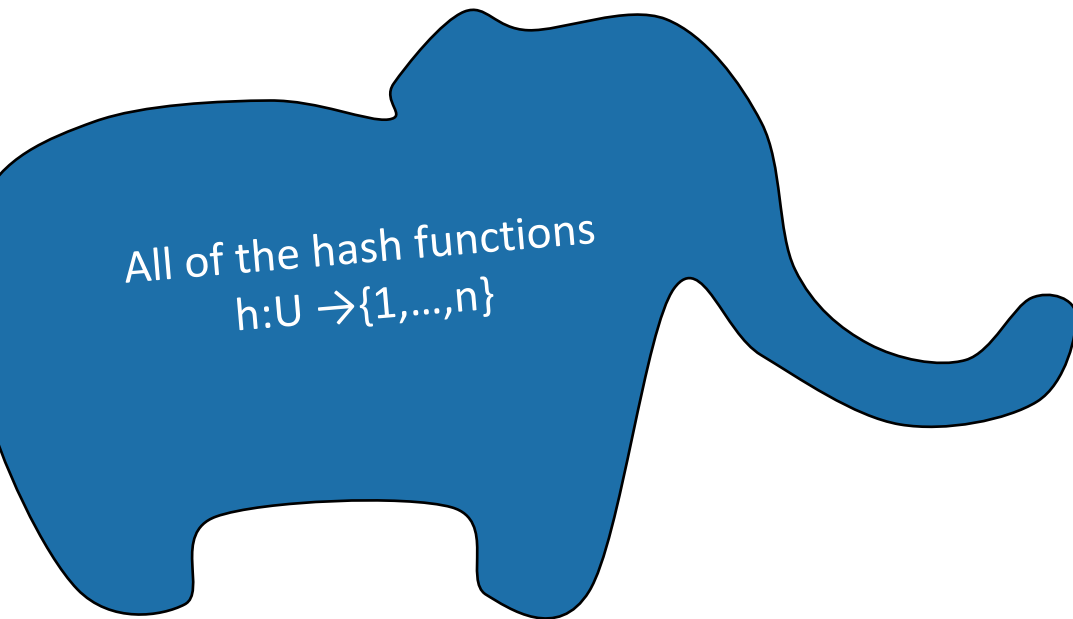
Maintain balance by stipulating that **black nodes** are balanced, and that there aren't too many **red nodes**.

It's just good sense!



Another way to store things

Hash tables!



Choose h randomly from a universal hash family.



It's better if the hash family is small!
Then it takes less space to store h .



The universe

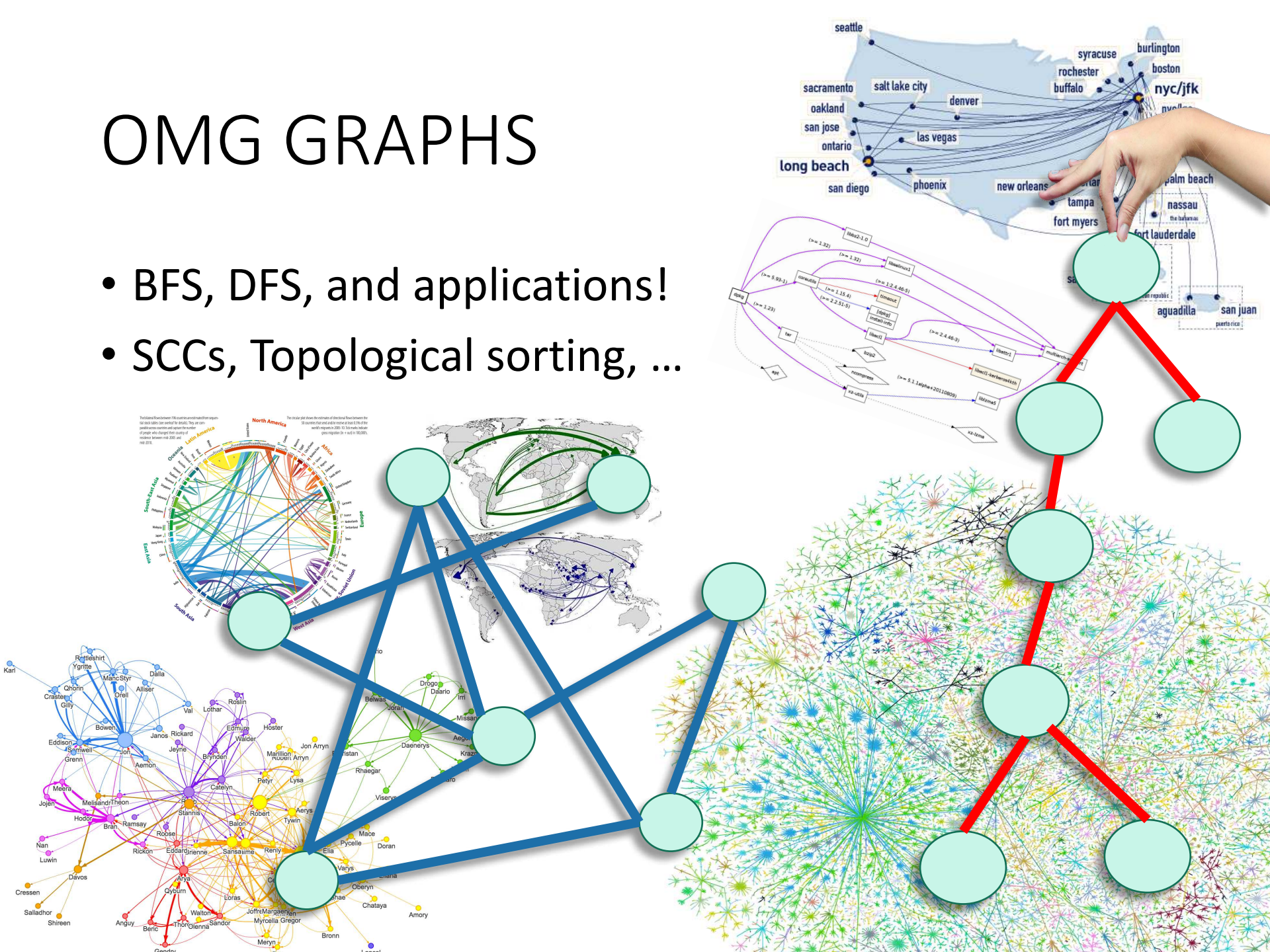
hash function h



Some buckets

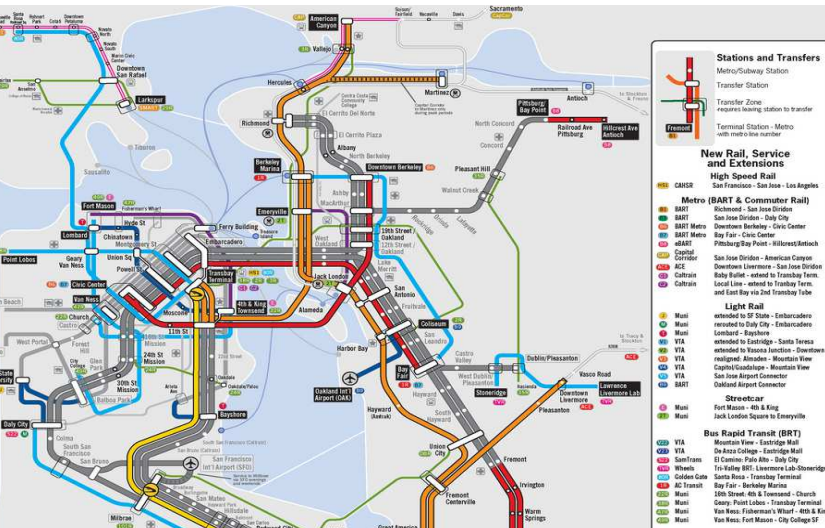
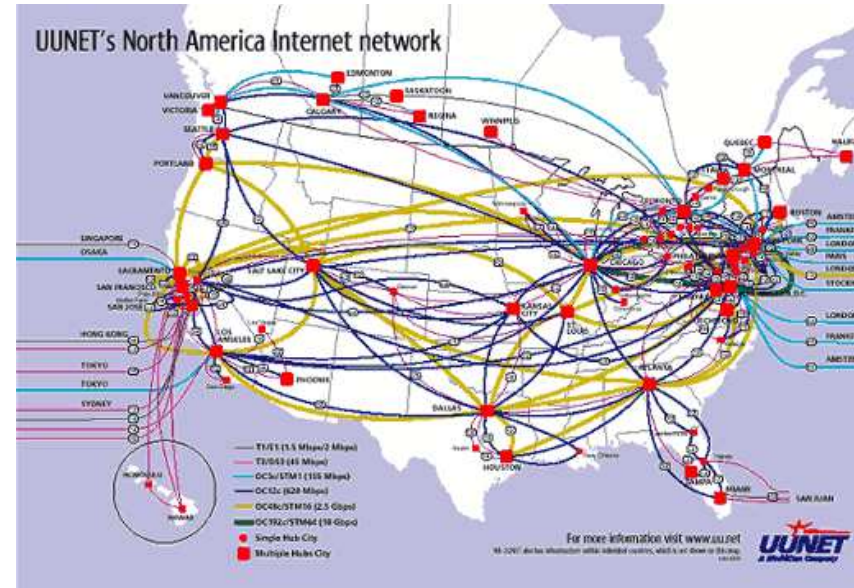
OMG GRAPHS

- BFS, DFS, and applications!
- SCCs, Topological sorting, ...



A fundamental graph problem: shortest paths

- Eg, transit planning, packet routing, ...
- Dijkstra!
- Bellman-Ford!
- Floyd-Warshall!



```

DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1 [AS0] 10.34.160.2 (10.34.160.2) 38.168 ms 31.272 ms 28.841 ms
 2 [AS0] cwa-vrtr.sunet (10.21.196.28) 33.769 ms 28.245 ms 24.373 ms
 3 [AS32] 171.66.2.229 (171.66.2.229) 24.468 ms 20.115 ms 23.223 ms
 4 [AS32] hpr-svl-rtr-vlan8.sunet (171.64.255.235) 24.644 ms 24.962 ms 17.453 ms
 5 [AS2152] hpr-svl-hpr2--stan-ge.cenic.net (137.164.27.161) 22.129 ms 4.902 ms 3.642 ms
 6 [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73) 12.125 ms 43.361 ms 32.3
 7 [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201) 40.174 ms 38.399 ms 34.499 ms
 8 [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28) 46.573 ms 23.926 ms 17
 9 [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31) 30.424 ms 25.770 ms 23.1
10 [AS0] et-4-0-0.4079.sdn-sw.denv.net.internet2.edu (162.252.70.8) 47.454 ms 57.273 ms 73.
11 [AS0] et-4-1-0.4079.rtsw.kans.net.internet2.edu (162.252.70.11) 70.825 ms 67.809 ms 62.1
12 [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206) 77.937 ms 57.421 ms 63.6
13 [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60) 77.682 ms 71.993 ms 73
14 [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65) 71.565 ms 74.988 ms 71.0
15 [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44) 154.926 ms 145.606 ms 145.872
16 [AS21320] ae0.mx1.lon2.uk.geant.net (62.40.98.79) 146.565 ms 146.604 ms 146.801 ms
17 [AS21320] ae0.mx1.par.fr.geant.net (62.40.98.77) 153.289 ms 184.995 ms 152.682 ms
18 [AS21320] ae2.mx1.gen.geant.net (62.40.98.153) 160.283 ms 160.104 ms 164.147 ms
19 [AS21320] swice1-100ge-0-3-0-1.switch.ch (62.40.124.22) 162.068 ms 160.595 ms 163.095 ms
20 [AS5559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94) 165.824 ms 164.216 ms 163.983 ms
21 [AS5559] swiez3-100ge-0-1-0-4.switch.ch (130.59.38.109) 164.269 ms 164.370 ms 163.929 ms
22 [AS5559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1) 164.082 ms 170.645 ms 165.372
23 [AS5559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169) 164.773 ms 165.193 ms 172.158 ms

```

Bellman-Ford and Floyd-Warshall
were examples of...

Dynamic Programming!

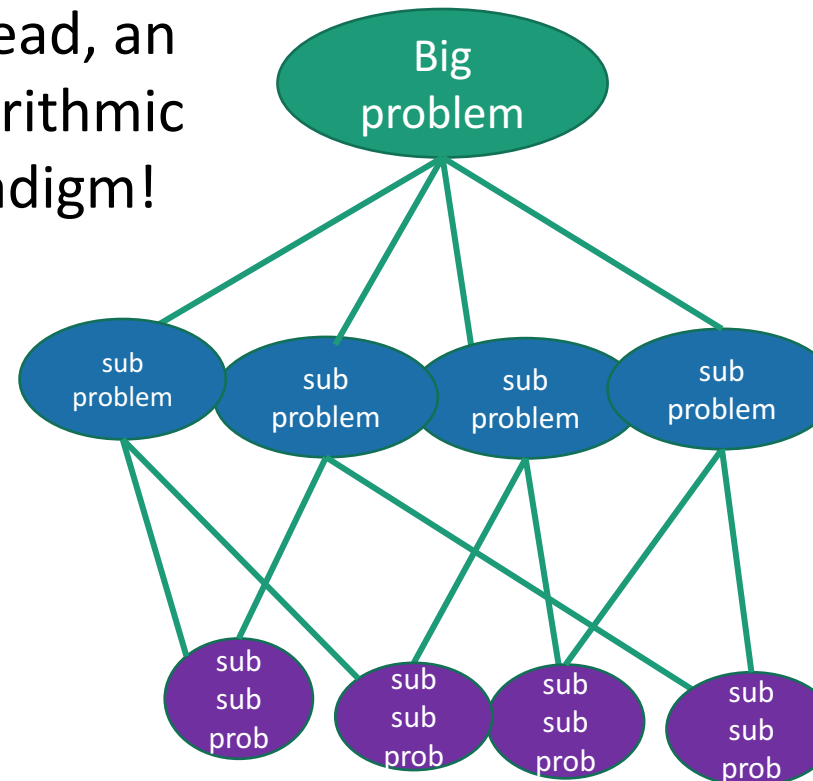
- Not programming in an action movie.



We saw many other
examples, including Longest
Common Subsequence and
Knapsack Problems.

Instead, an
algorithmic
paradigm!

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Steps 3-5:** Use dynamic programming: fill in a table to find the answer!



Sometimes we can take even better advantage of optimal substructure...with Greedy algorithms

- Make a series of choices, and commit!



- Intuitively we want to show that our greedy choices never rule out success.
- Rigorously, we usually analyzed these by induction.

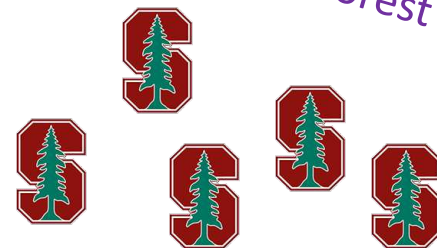
- Examples!

- Activity Selection
- Job Scheduling
- Huffman Coding
- **Minimum Spanning Trees**

*Prim's algorithm:
greedily grow a tree*



*Kruskal's algorithm:
greedily grow a forest*

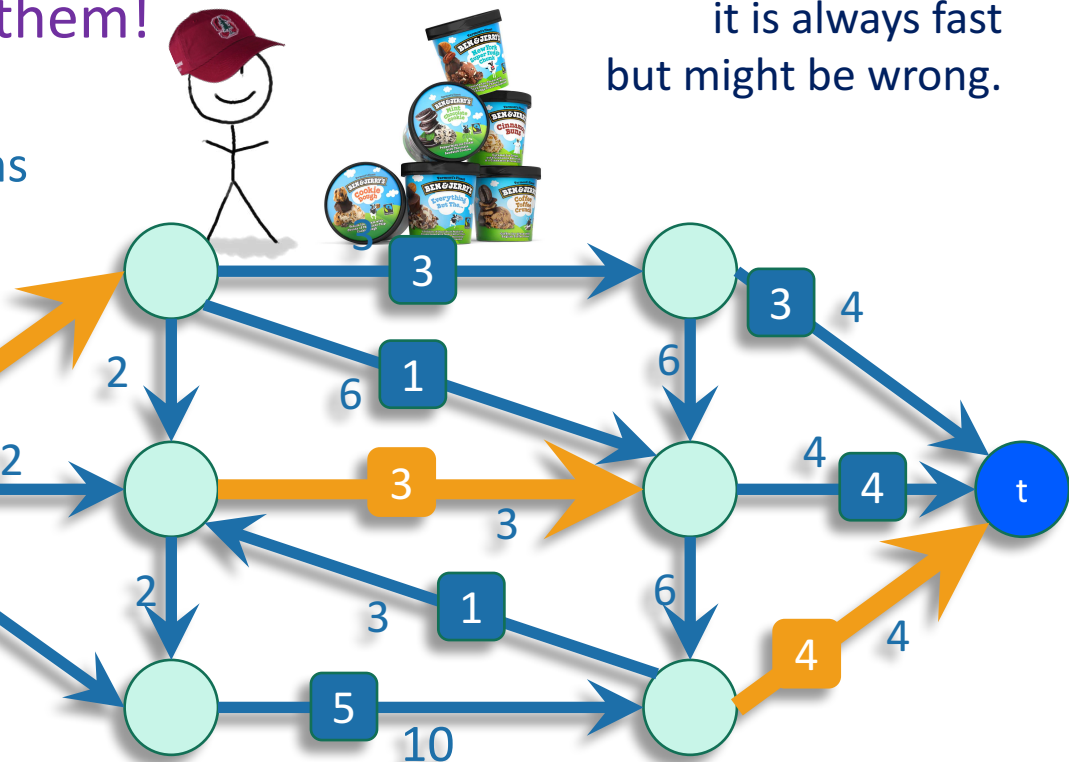
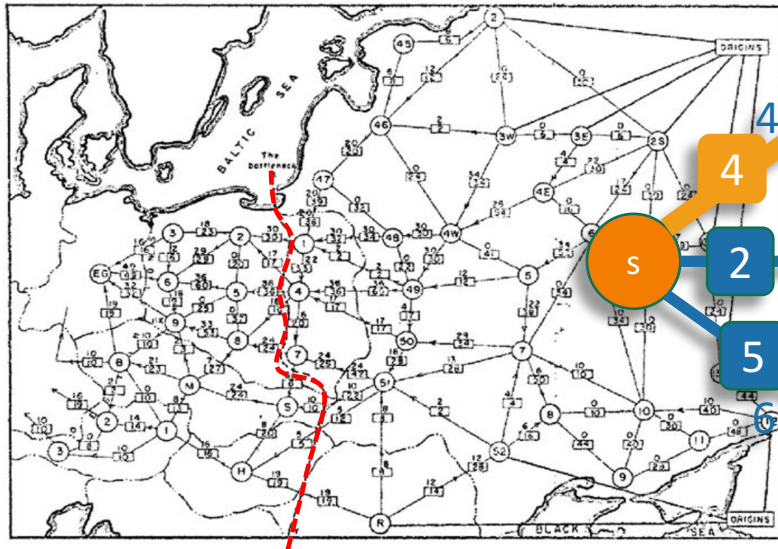


Cuts and flows

- Global minimum cut:
 - Karger's algorithm!
 - minimum s-t cut:
 - is the same as maximum s-t flow!
 - Ford-Fulkerson can find them!
 - useful for routing
 - also assignment problems
- 



Karger's algorithm is a **Monte-Carlo algorithm**:
it is always fast
but might be wrong.



And now we're here



What have we learned?

- A few algorithm design paradigms:
 - Divide and conquer, Dynamic Programming, Greedy
- A few analysis tools:
 - Worst-case analysis, asymptotic analysis, recurrence relations, probability tricks, proofs by induction
- A few common objects:
 - Graphs, arrays, trees, hash functions
- A LOT of examples!



What have we learned?

We've filled out a toolbox

- Tons of examples give us intuition about what algorithmic techniques might work when.
- The technical skills make sure our intuition works out.



But there's lots more out there



- What's next???

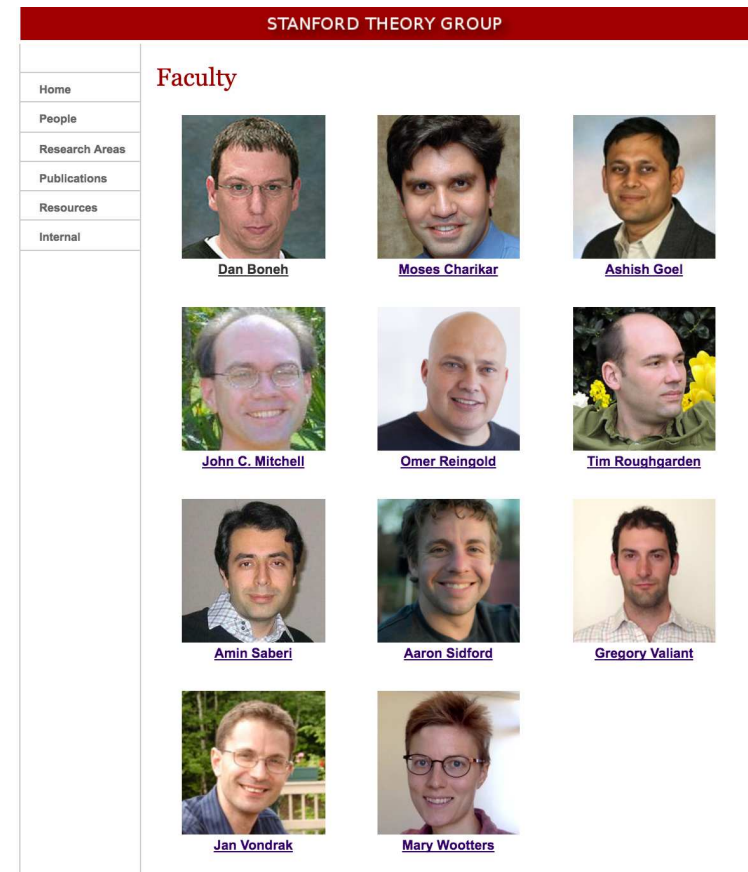
A taste of what's to come

- CS154 – Introduction to Complexity
- CS167 – Readings in Algorithms
- CS168 – The Modern Algorithmic Toolbox
- MS&E 212 – Combinatorial Optimization
- CS250 – Error Correcting Codes
- CS254 – Computational Complexity
- CS255 – Introduction to Cryptography
- CS261 – A Second Course in Algorithms
- CS264 – Beyond Worst-Case Analysis
- CS265 – Randomized Algorithms
- CS269 – Incentives in Computer Science
- EE364A/B – Convex Optimization I and II

...and many many more
upper-level topics courses!

findSomeTheoryCourses():

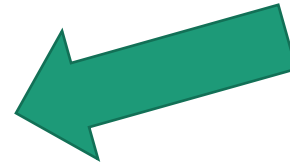
- go to theory.stanford.edu
- Click on “People”
- Look at what we’re teaching!



Today

A few gems

- Linear programming



This will be pretty fluffy,
without much detail –
take more CS theory
classes for more detail!

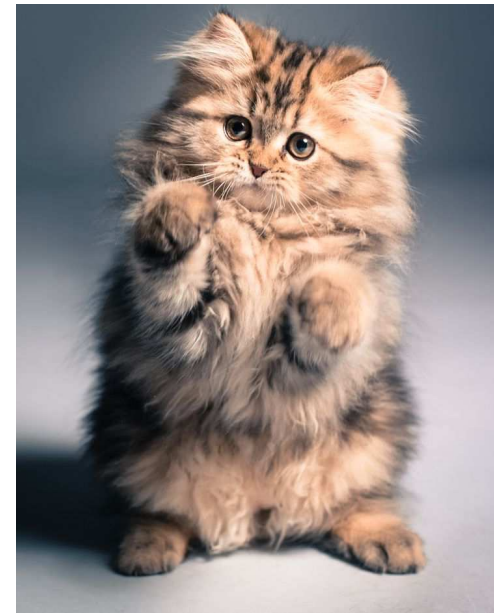
- Random projections



- Low-degree polynomials



**NOTHING AFTER THIS POINT
WILL BE ON THE FINAL EXAM**



Linear Programming

- This is a fancy name for optimizing a **linear function** subject to **linear constraints**.
- For example:

Maximize

$$x + y$$

subject to

$$x \geq 0$$

$$y \geq 0$$

$$4x + y \leq 2$$

$$x + 2y \leq 1$$

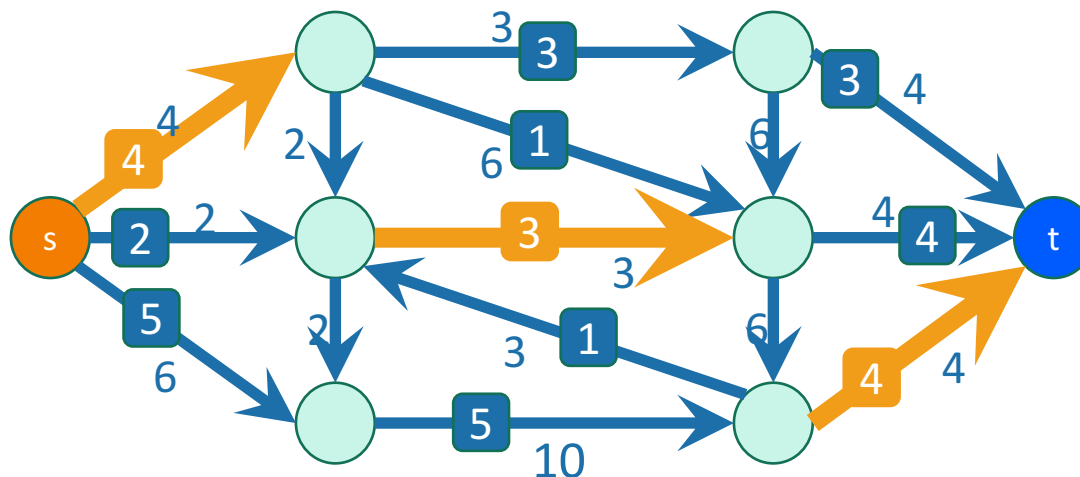
- It turns out to be an **extremely general** problem.

Actually we just saw it on Monday

Maximize
the sum of the
flows leaving s

subject to

- None of the flows are bigger than the edge capacities
- At every vertex, stuff going in = stuff going out.



Another example, in machine learning

Support Vector Machines

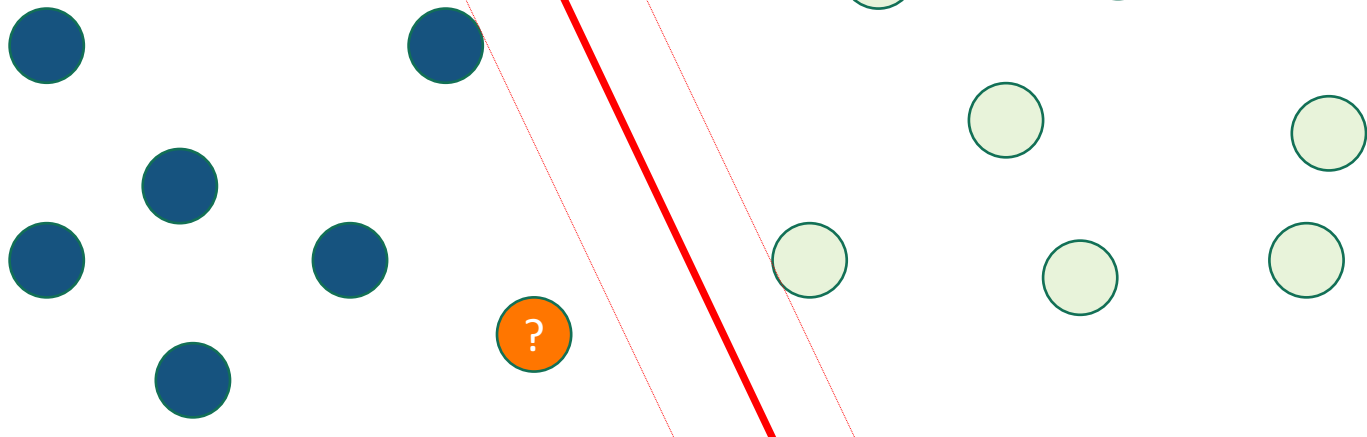
Maximize
the margin

subject to

This can be written
in a linear way

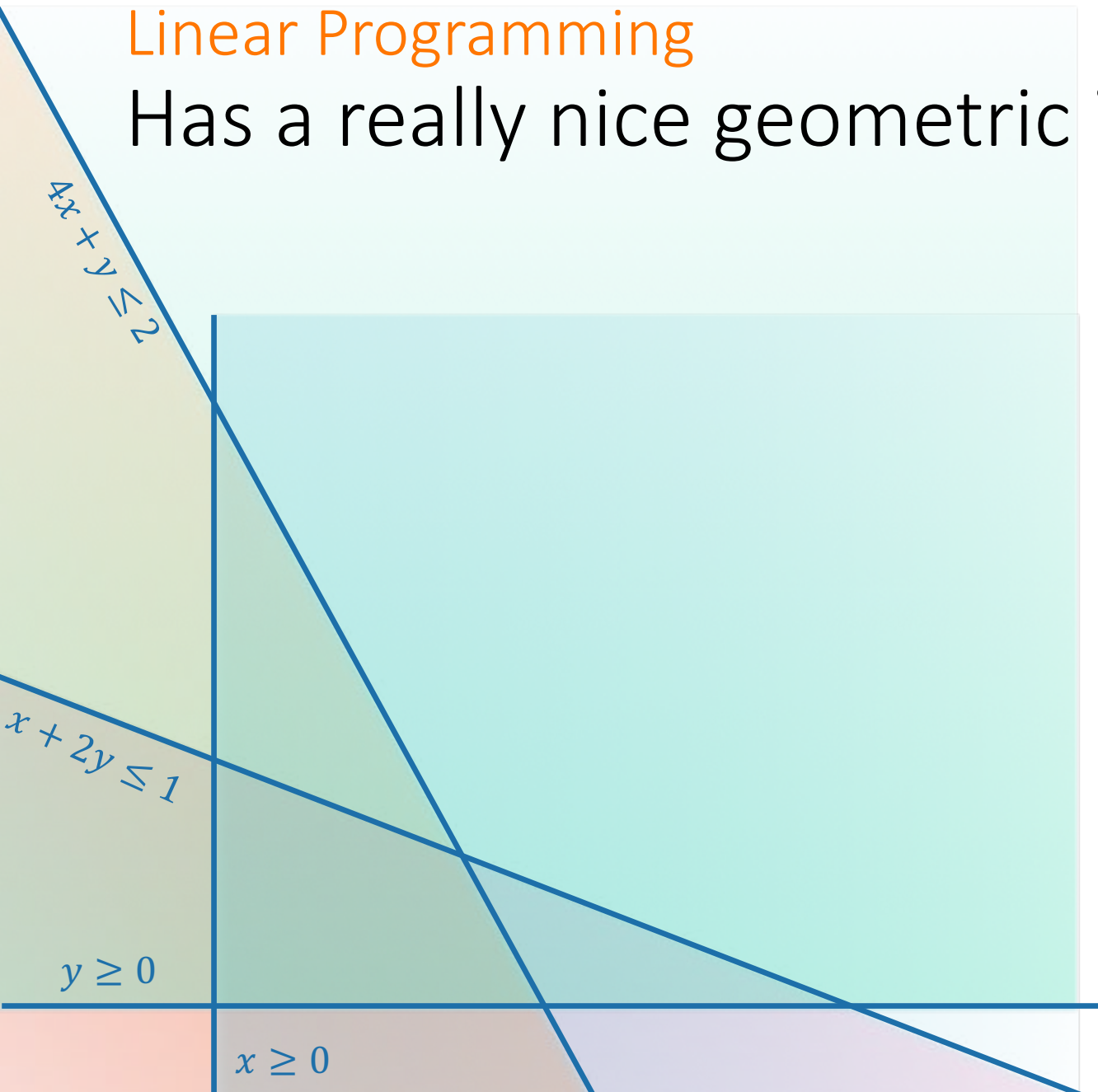
all of the points are
on the correct side
of the line.

These are just
linear inequalities



Linear Programming

Has a really nice geometric intuition



Maximize

$$x + y$$

subject to

$$x \geq 0$$

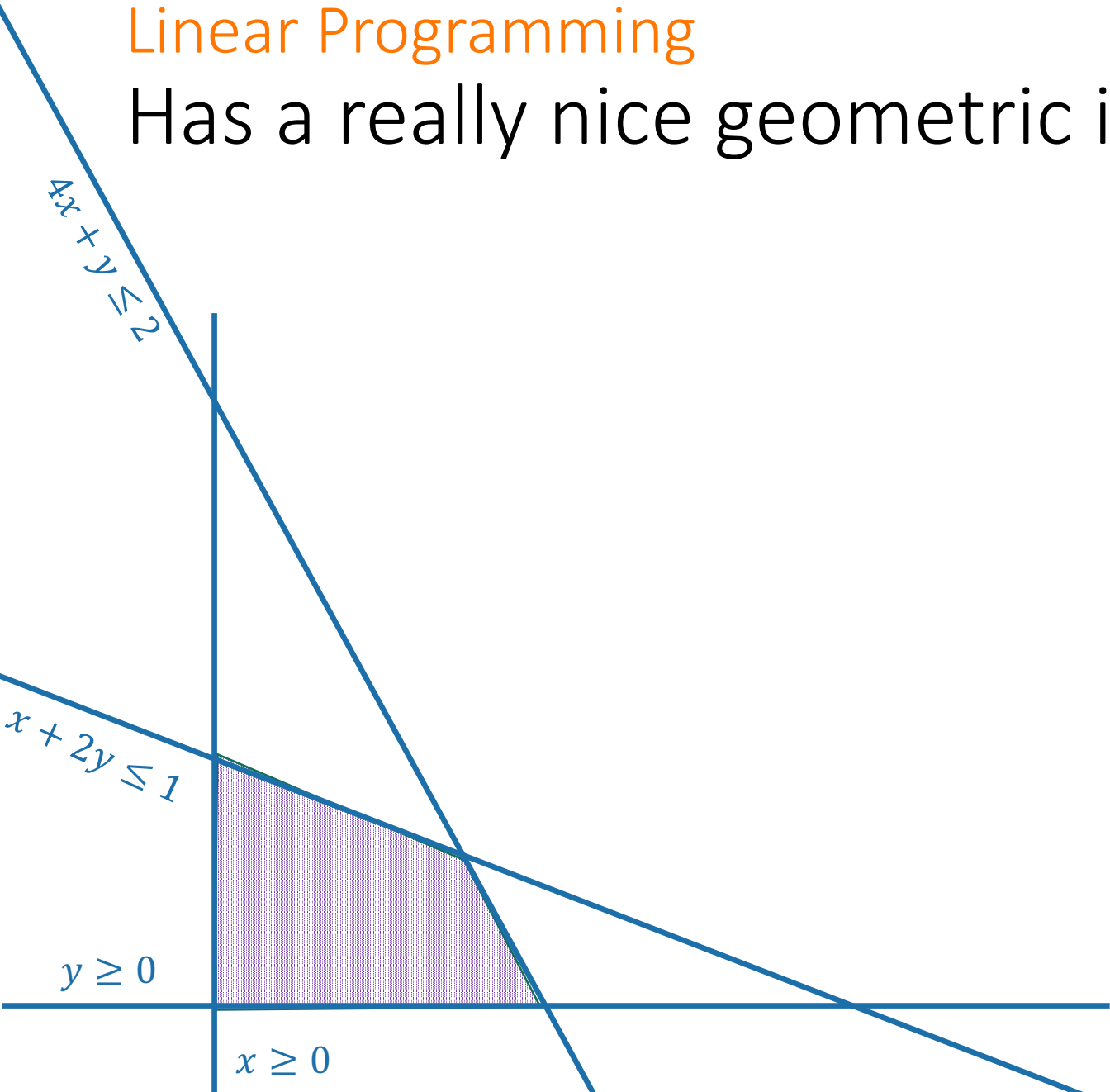
$$y \geq 0$$

$$4x + y \leq 2$$

$$x + 2y \leq 1$$

Linear Programming

Has a really nice geometric intuition



Maximize

$$x + y$$

subject to

$$x \geq 0$$

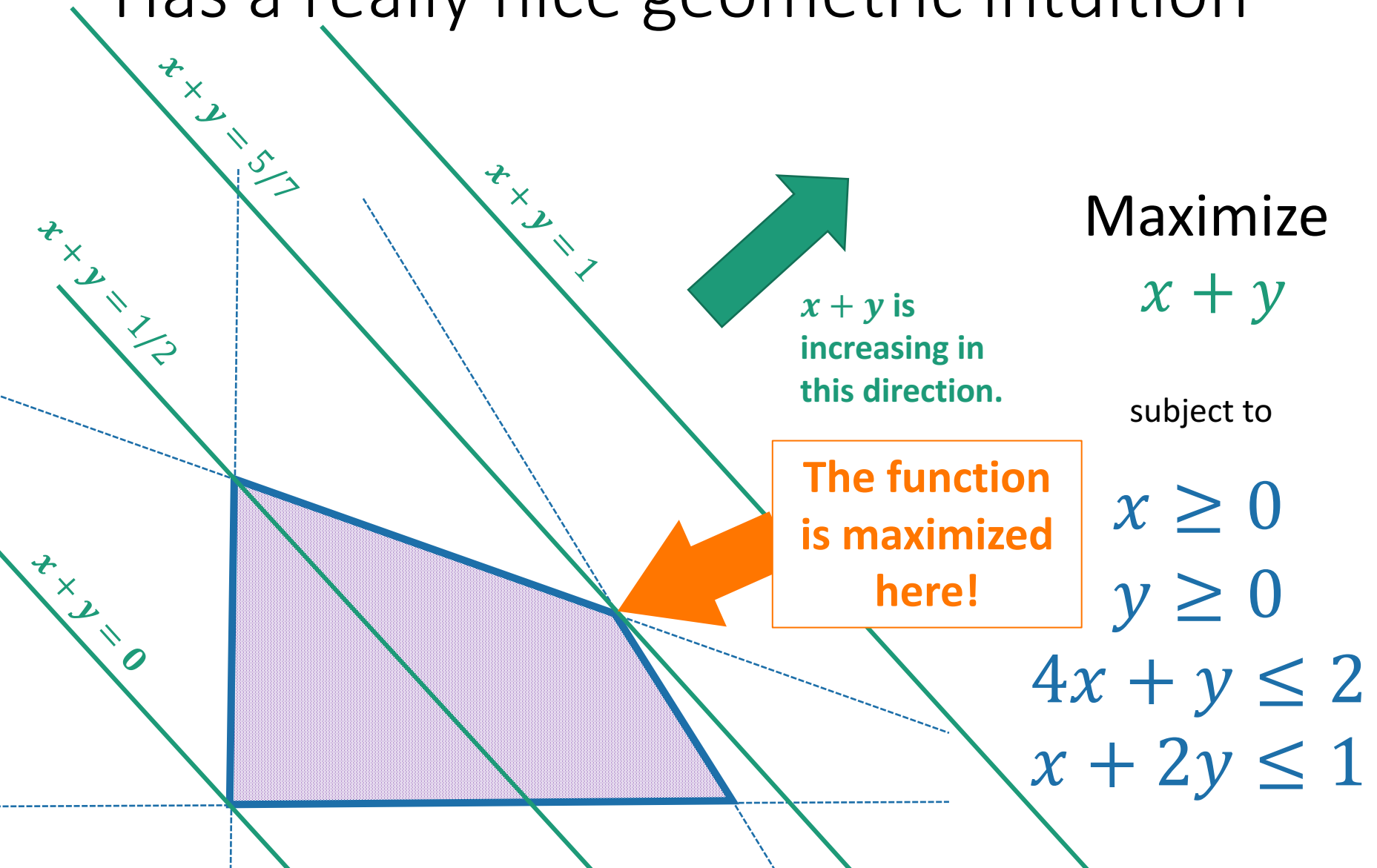
$$y \geq 0$$

$$4x + y \leq 2$$

$$x + 2y \leq 1$$

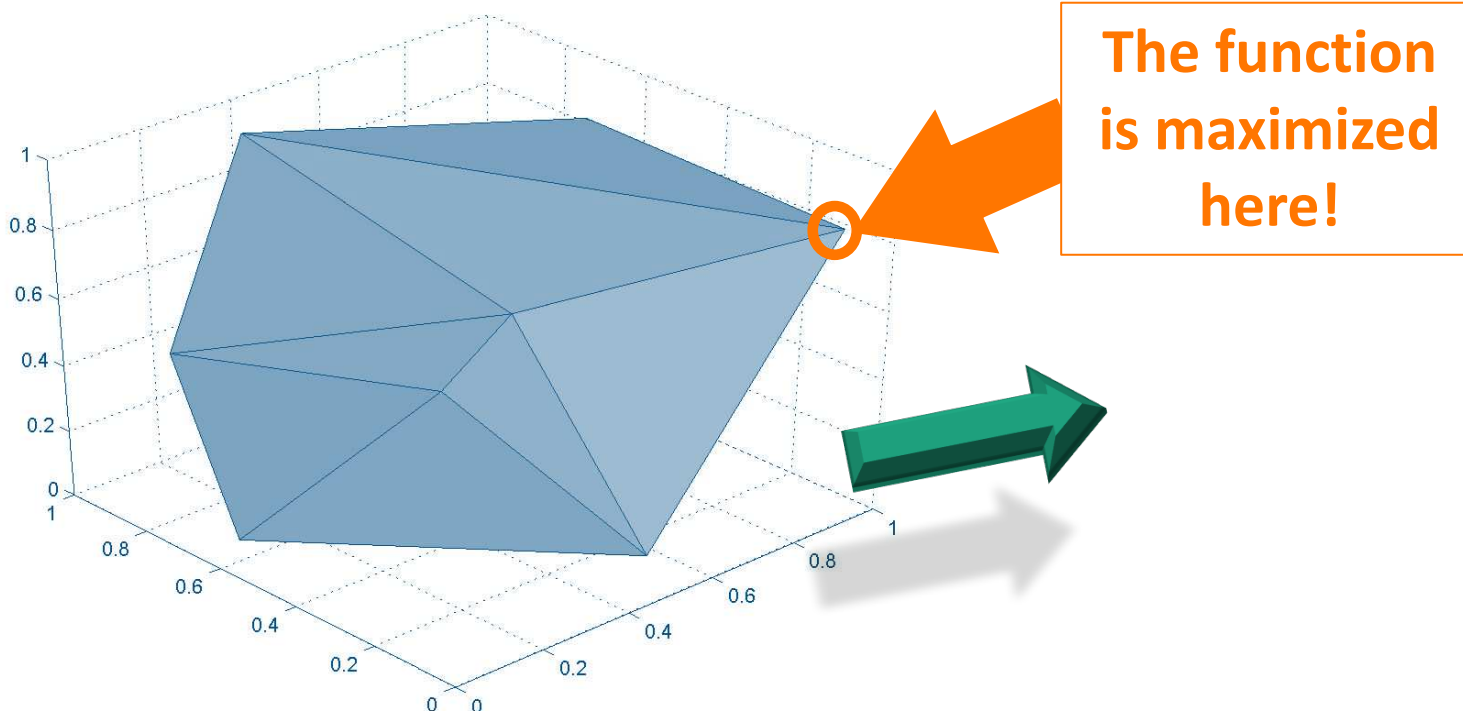
Linear Programming

Has a really nice geometric intuition



In general

- The constraints define a **polytope**
- The function defines a **direction**
- We just want to find the vertex that is **furthest in that direction**.



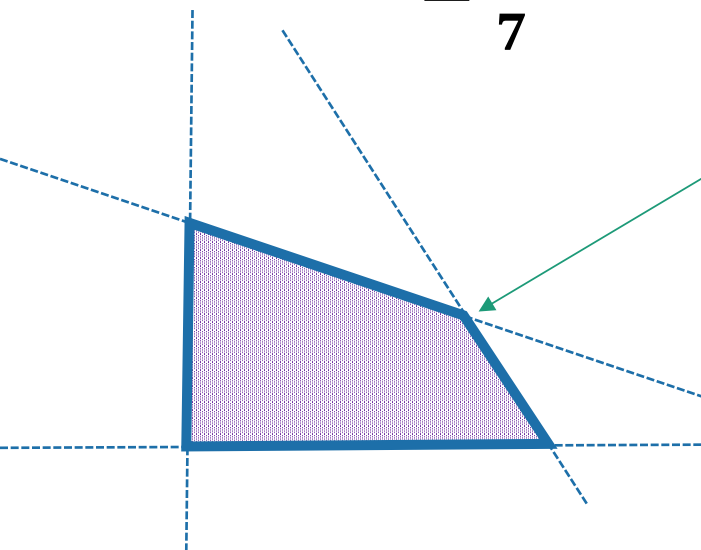
Duality

How do we know we have an optimal solution?

I claim that the optimum is 5/7.

Proof: say x and y satisfy the constraints.

- $x + y = \frac{1}{7}(4x + y) + \frac{3}{7}(x + 2y)$
- $\leq \frac{1}{7} \cdot 2 + \frac{3}{7} \cdot 1$
- $= \frac{5}{7}$



You can check this point has value 5/7...but how would we prove it's optimal other than by eyeballing it?

Maximize
 $x + y$

subject to

$$x \geq 0$$

$$y \geq 0$$

$$4x + y \leq 2$$

$$x + 2y \leq 1$$

cute, but

How did you come up with $1/7, 3/7$?

I claim that the optimum is $5/7$.

Proof: say x and y satisfy the constraints.

- $x + y \leq (4x + y) + (x + 2y)$
- $\leq \cdot 2 + 1$
- $=$

- I want to choose things to put here
- So that I minimize this
- Subject to these things

Maximize
 $x + y$

subject to

$$x \geq 0$$

$$y \geq 0$$

$$4x + y \leq 2$$

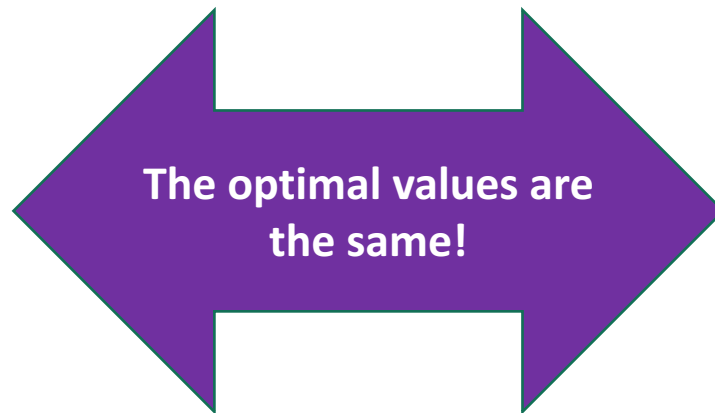
$$x + 2y \leq 1$$

That's a linear program!

- How did I find those special values $1/7, 3/7$?
- I solved some linear program. Minimize the upper bound you get, subject to the proof working.
- It's called the dual program.

Maximize stuff
subject to stuff

Primal



Minimize other stuff
subject to other stuff

Dual

We've actually already seen this too

The Min-Cut Max-Flow Theorem!

Maximize the
sum of the
flows leaving s

s.t.

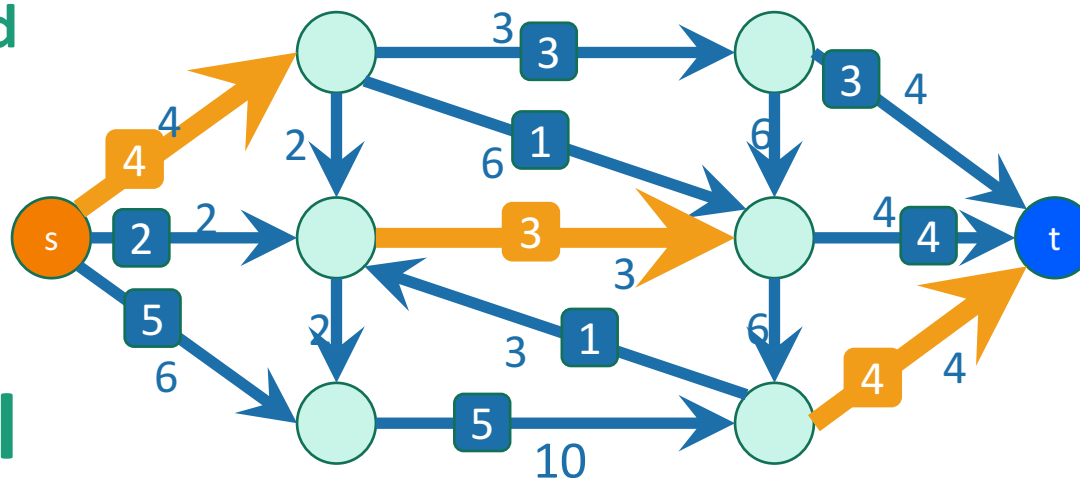
All the flow
constraints are
satisfied

The optimal values are
the same!

Minimize the sum
of the capacities
on a cut

s.t.

it's a legit cut

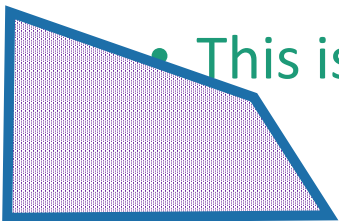


Primal

Dual

LPs and Duality are really powerful

- This **general phenomenon** shows up all over the place
 - Min-Cut Max-Flow is a special case.
- Duality helps us reason about an optimization problem
 - The dual provides a **certificate** that we've solved the primal.
 - eg, if you have a cut and a flow with the same value, you must have found a max flow and a min cut.
- We can solve LPs quickly!
 - For example, by intelligently bouncing around the vertices of the feasible region.
 - This is an **extremely powerful algorithmic primitive**.



Today

A few gems

- Linear programming



- Random projections



- Low-degree polynomials



One of my favorite tricks

Take a random projection and hope for the best.

High-dimensional set of points

For example, each data point is a vector
(age, height, shoe size, ...)

Their shadow is a projection onto the ground.

Choose a random subspace to project onto instead of the ground.

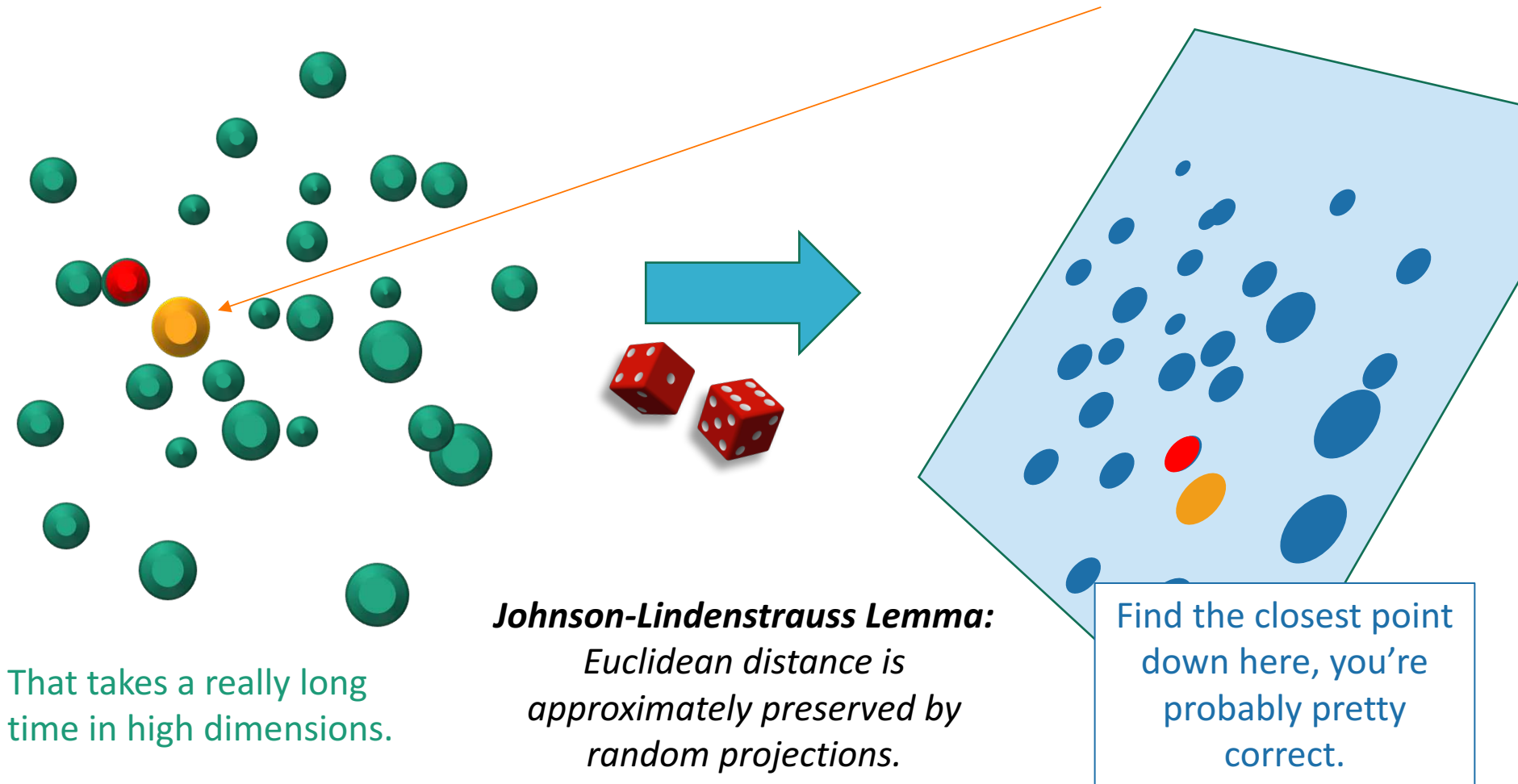


Why would we do this?

- High dimensional data takes a long time to process.
- Low dimensional data can be processed quickly.
- **"THEOREM"**: Random projections approximately preserve properties of data that you care about.

Example: nearest neighbors

- I want to find which point is closest to **this one**.



Another example: Compressed Sensing

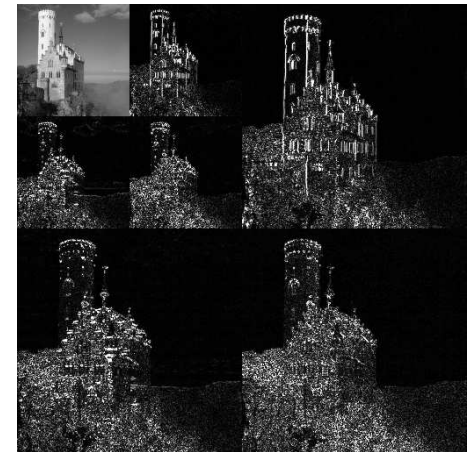
- Start with a **sparse** vector
 - Mostly zero or close to zero

(**5** , 0 , 0 , 0 , 0 , 0.01 , 0.01 , **5.8** , **32** , **14** , 0 , 0 , 0 , **12** , 0 , 0 , **5** , 0 , .03)

- For example:



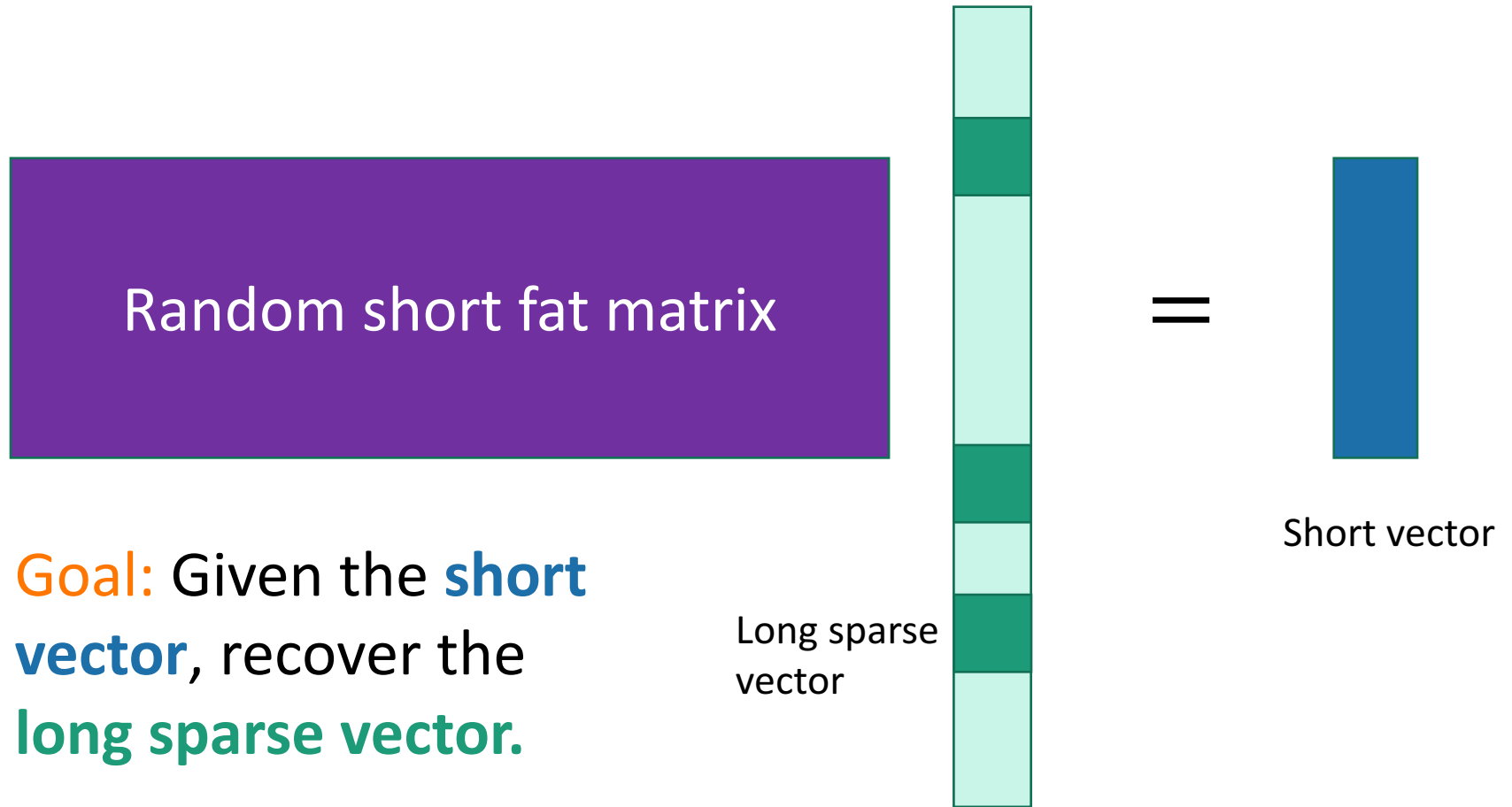
This image is sparse



This image is sparse after I
take a wavelet transform.

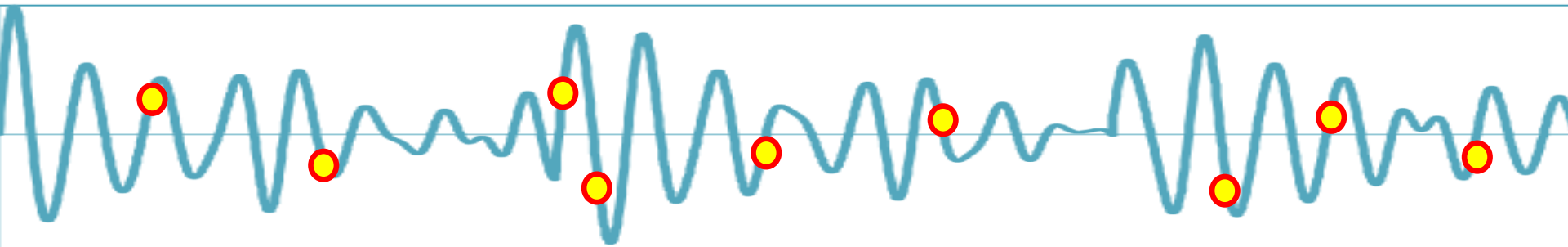
Compressed sensing continued

- Take a random projection of that sparse vector:



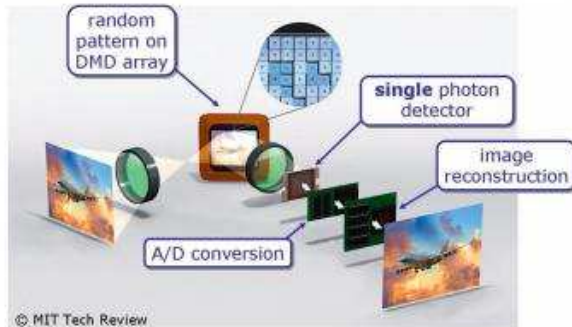
Why would I want to do that?

- Image compression and signal processing
- Especially when you **never have space to store the whole sparse vector to begin with.**



Randomly sampling (in the time domain) a signal that is sparse in the Fourier domain.

Random measurements in an fMRI means you spend less time inside an fMRI

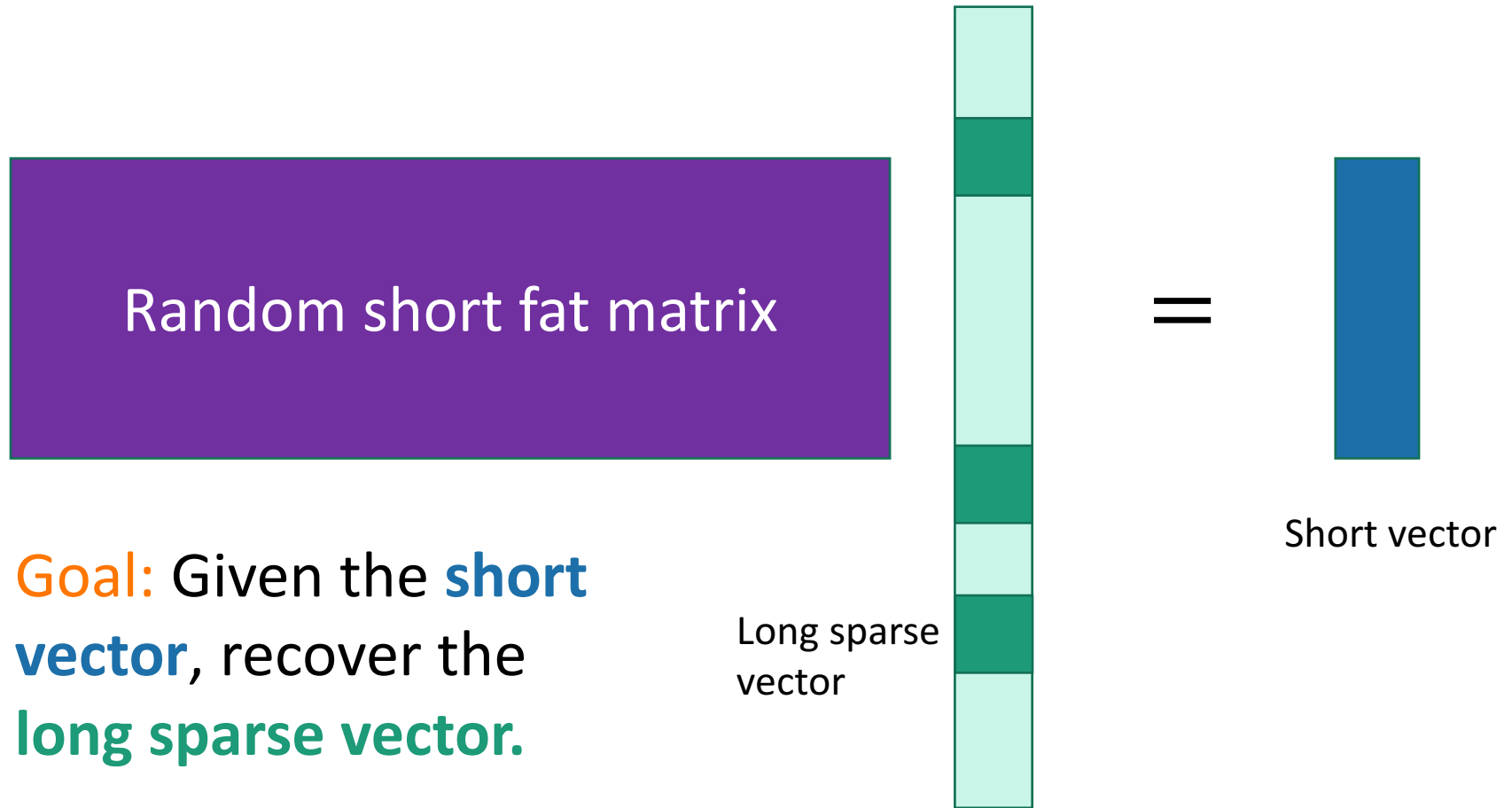


A “single pixel camera” is a thing.



But why should this be possible?

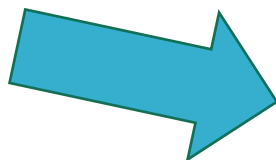
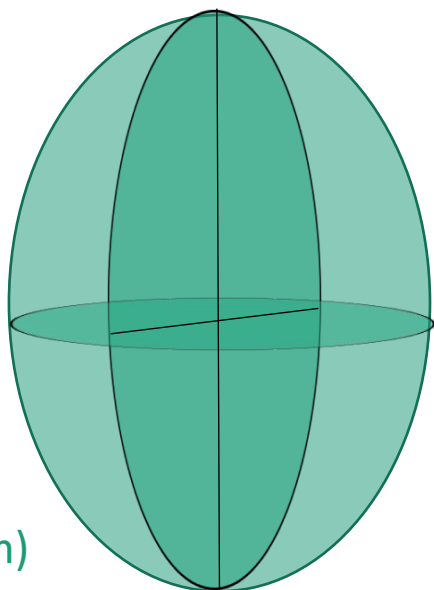
- There are tons of long vectors that map to the short vector!



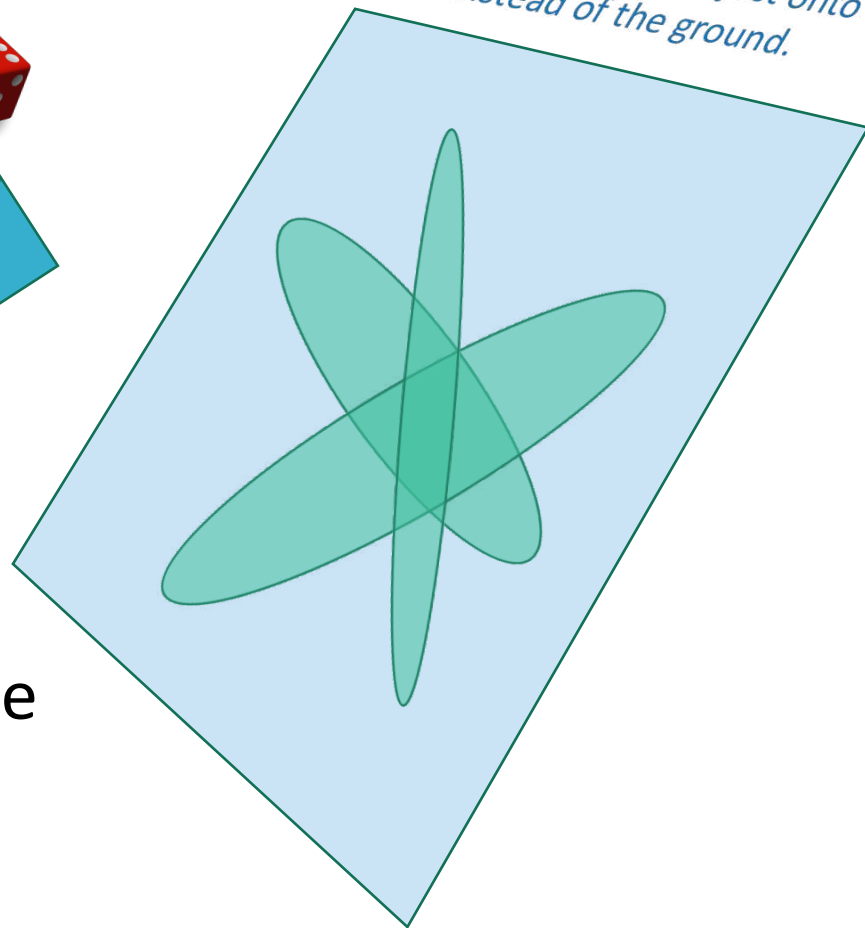
Back to the geometry

All of the
sparse
vectors

(Infinitely
many of them)



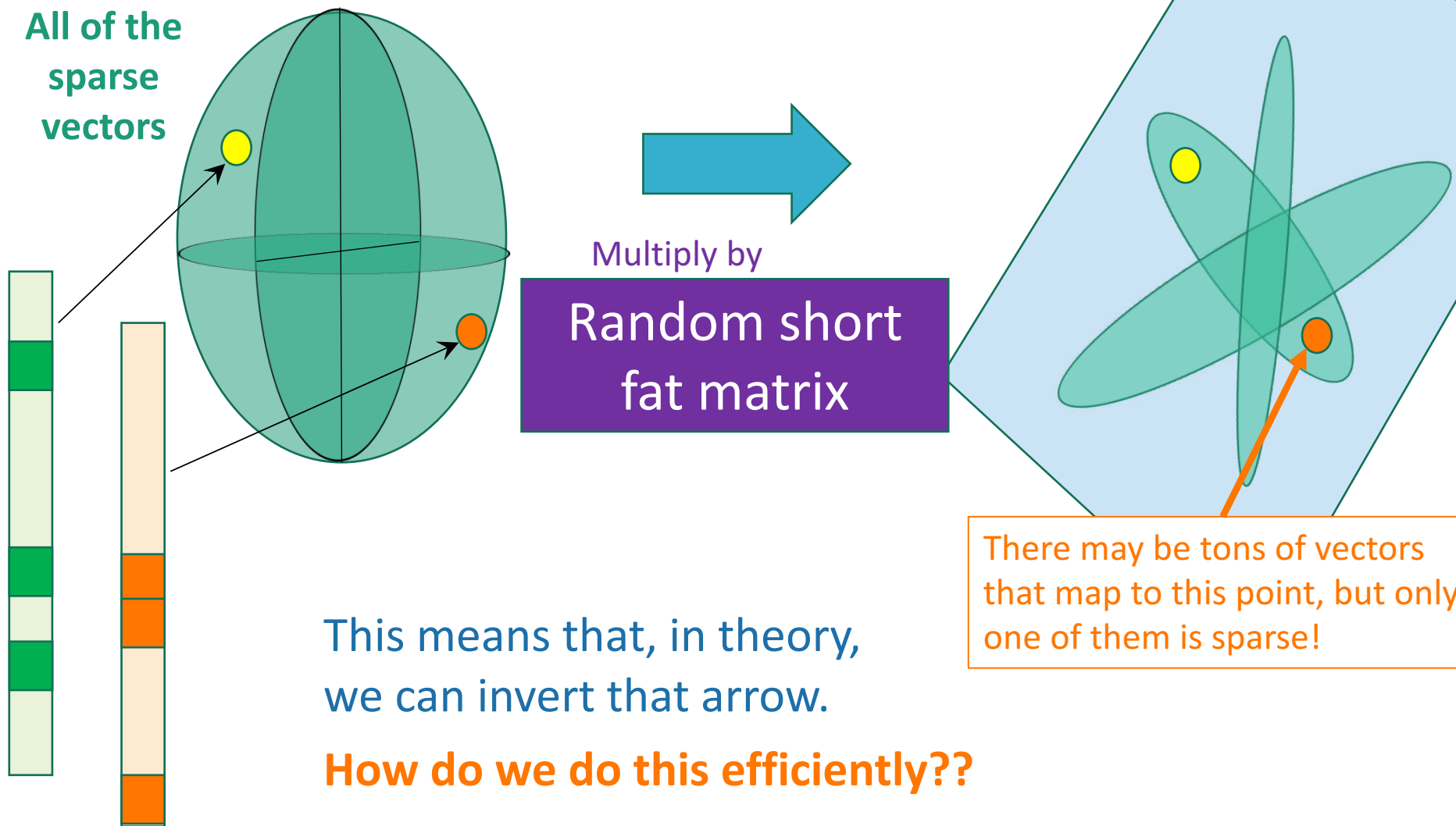
*Choose a random
subspace to project onto
instead of the ground.*



Theorem:

random projections preserve the
geometry of sparse vectors too.

If we don't care about algorithms,
that's more than enough.



Goal: Given the **short vector**,
recover the **long sparse vector**.

An efficient algorithm?

What we'd like to do is:

Minimize number of
nonzero entries in x

This norm is the sum
of the absolute values
of the entries of x

This isn't a
nice function

Instead:

Minimize $\|x\|_1$

s.t.

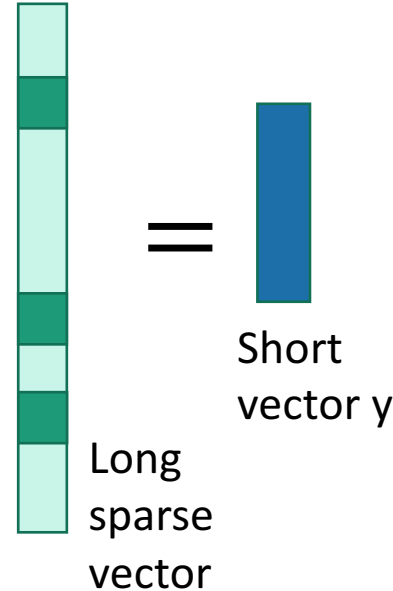
$$Ax = y$$

Problem: I don't know
how to do that efficiently!

s.t.

$$Ax = y$$

Random short
fat matrix A



- It turns out that because the geometry of sparse vectors is preserved, this optimization problem **also works**.
- We can use **linear programming** to solve this quickly!

Today

A few gems

- Linear programming



- Random projections



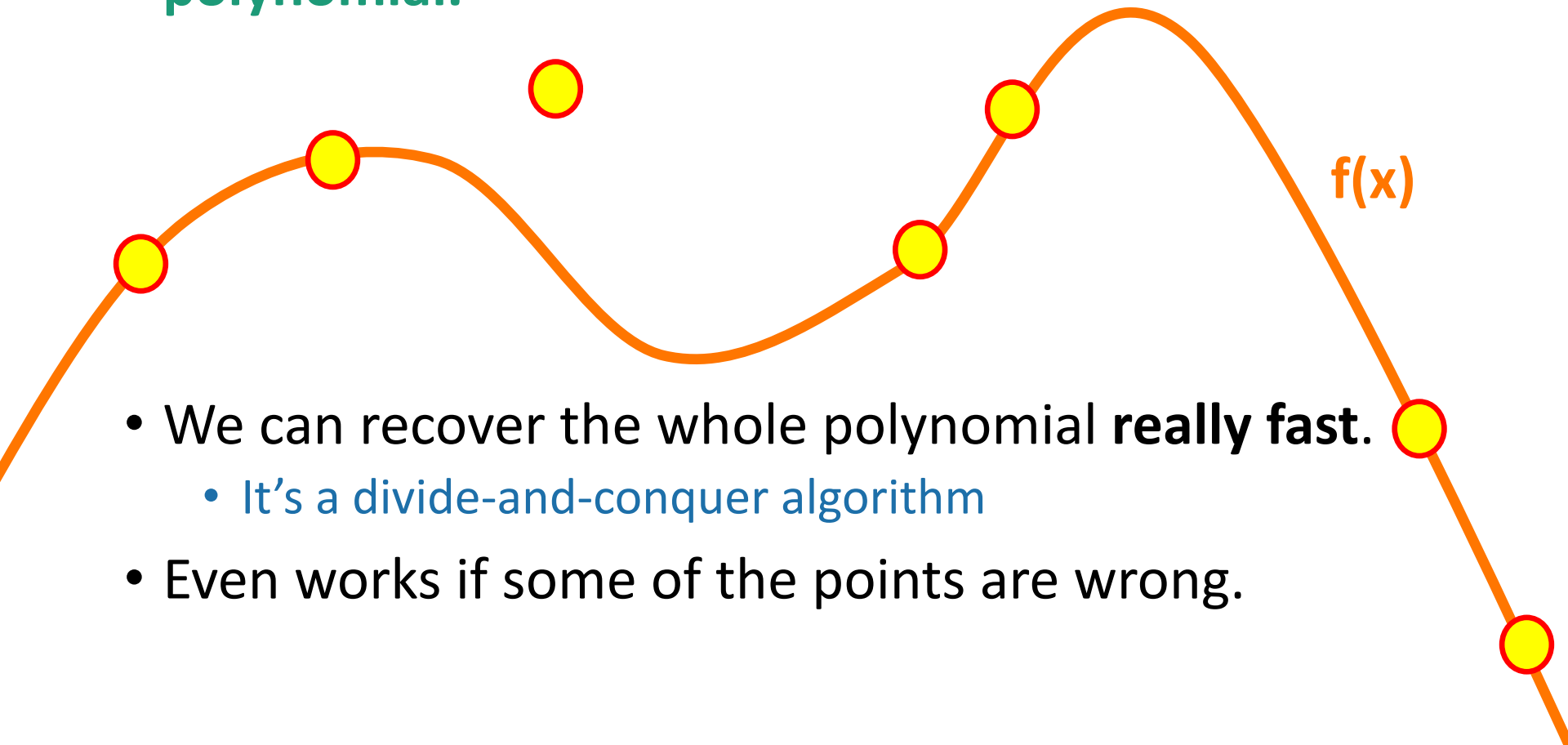
- Low-degree polynomials



Another of my favorite tricks

Polynomial interpolation

- Say we have a few evaluation points of a **low-degree polynomial**.



- We can recover the whole polynomial **really fast**.
 - It's a divide-and-conquer algorithm
- Even works if some of the points are wrong.

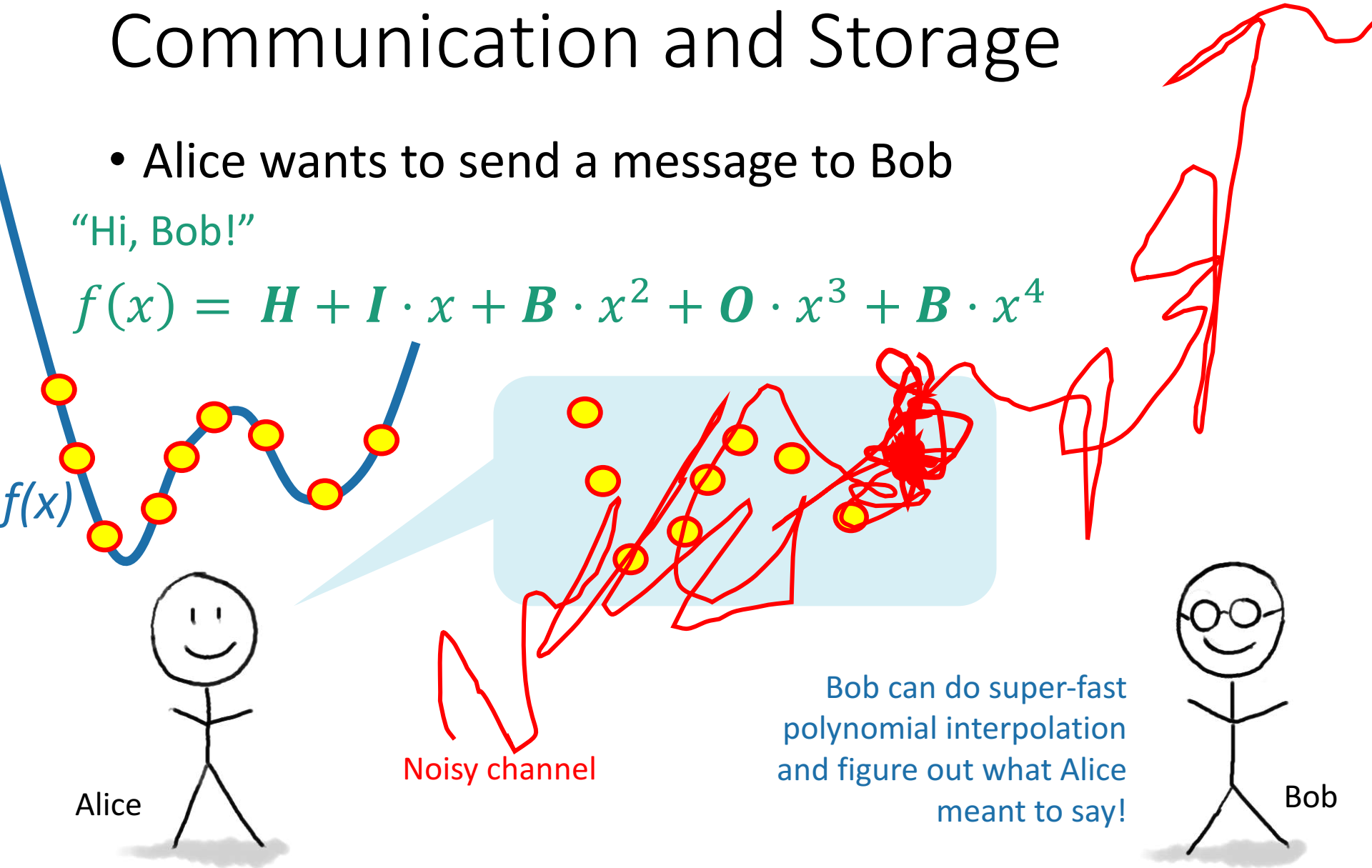
One application:

Communication and Storage

- Alice wants to send a message to Bob

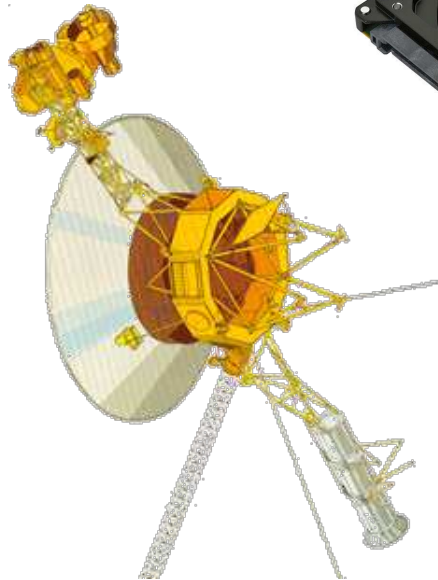
"Hi, Bob!"

$$f(x) = H + I \cdot x + B \cdot x^2 + O \cdot x^3 + B \cdot x^4$$



This is actually used in practice

- It's called "Reed-Solomon Encoding"



Another application:

Designing “random” projections that are better than random



Random short fat matrix

The matrix that treats the big long vector as Alice’s message polynomial and evaluates it REAL FAST at random points.

- This is still “random enough” to make the LP solution work.
- It is much more efficient to manipulate and store!

Today

A few gems

- Linear programming



- Random projections



- Low-degree polynomials



To learn more:

CS168, CS261, ...

CS168, CS261,
CS264, CS265, ...

CS168, CS250, ...

What have we learned?

CS161



Tons more cool
algorithms stuff!

To see more...

- Take more classes!
- Come hang out with the theory group!
 - Theory lunch, Thursdays at noon
 - Theory seminar, Thursdays at 4:15
 - (Not during the summer, but next fall!)

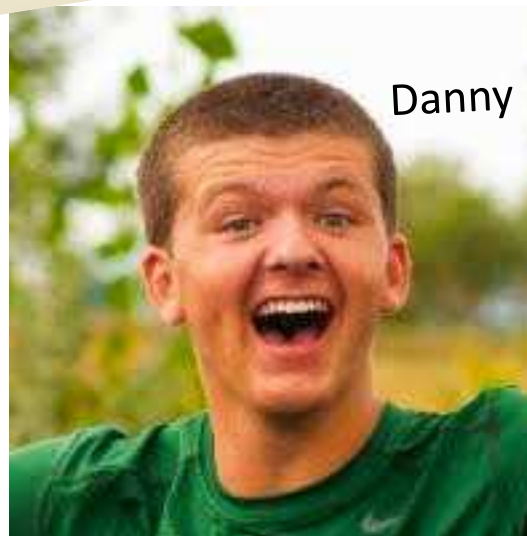
theory.stanford.edu

Stanford theory group:
We are very friendly.



A few final messages...

1. Thanks to the TAs!!!
tell them you appreciate them!



2.



THANKS
to you!!!!!!

